BP 神经网络在 mnist 数据集中的应用

张阳

2022 年 6 月 7 日

摘要

本文首先介绍了神经网络的基本知识,重点说明了感知器和 S 神经元;随后给出了一种求解神经网络的方法:反向传播算法,并给出了数学上的证明。完成准备工作后,将算法应用到了 minst 手写数学数据集中。为了进一步提升运算速度和改善识别结果,使用了 SGD 方法,最终数字识别正确率达到了 95% 以上。在文章的最后,说明了神经网络的难以训练性。

目录

1 引言

近年来,机器学习方兴未艾。2016 年,alphago 击败李世石,宣告着围棋这一古老的人类运动人类已经彻底败给了计算机,再一次掀起了广大民众对机器学习的讨论。导航,社交媒体平台,图像识别,情感分析,机器学习的强大效力一次又一次地刷新着我们的认知。在一边赞叹机器学习强大的同时,也有越来越多的学者参与到了它的建设之中,可以说,机器学习就是现在科学界最热门,最激动人心的方向。

神经网络是一种用计算机模拟人脑思维过程的算法,属于机器学习的一种。神经网络的功能很多,在此文中,我们特别以 mnist 数据集为例来制作一个字体识别的神经网络。

2 生成一个神经网络

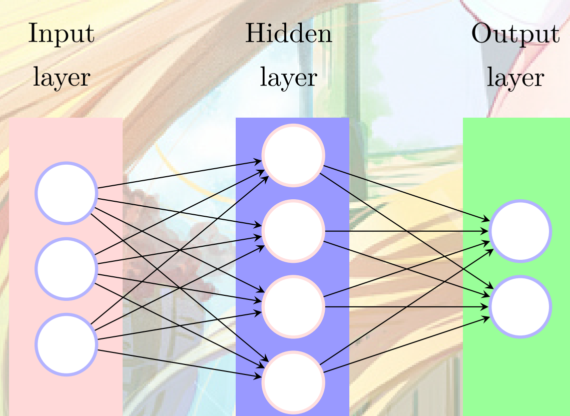
2.1 神经网络结构

在人类的大脑中,有着数不清的神经元进行着复杂的化学反应,突触起到了神经元之间的连接作用。这些神经元和突触错综复杂地结合在一起,赋予了我们思考的能力。

受制于硬件水平,我们的神经网络无法达到人类大脑的复杂度,不过好在,我们也不需要让神经网络来承担一个人的所有工作,我们只需要它能够完成一些特殊的任务就好了。

首先,神经网络应当由一些神经元来接受信息,这些神经元构成输入层 (input layer);需要一些神经元来运算,处理信息,这些神经元构成隐藏层 (hidden layer);还有一些神经元来输出信息,构成输出层 (output layer)。神经元当然不能独立工作,因此,我们需要将神经元连接起来,为了简化复杂度,我们认为同层之间的神经元不会由之间的影响,只有非同层的神经元才会互相影响。

于是我们可以简单画一个神经网络来做例子:



2.2 神经网络运算方法

首先,我们先介绍感知器:

一个感知器会接受几个二进制输入,$x_1, x_2, \cdots, x_n$,并产生一个二进制输出。具体来说就是

$$f(x_1, x_2, \cdots, x_n) = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases} \tag{2.1}$$

其中,w_i 表示权重。直观上理解,如果感知器受到的刺激超过了某一个预设的阈值,就输出 1;小于这个阈值,就输出 0。

为了简化表达式,我们把 (2.1) 式写成向量的形式。[1]

$$f(\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{x}\cdot\mathbf{w}+b<0 \\ 1 & \text{if } \mathbf{x}\cdot\mathbf{w}+b>0 \end{cases} \tag{2.2}$$

其中 $\mathbf{x}=(x_1,x_2,\cdots,x_n)$, $\mathbf{w}=(w_1,w_2,\cdots,x_n)$, $\mathbf{x}\cdot\mathbf{w}=\sum x_i w_i$, b 为预设的阈值。

如果令 $z=\mathbf{x}\cdot\mathbf{w}$,上式可以进一步化简为:

$$f(\mathbf{x}) = \begin{cases} 0 & \text{if } z<0 \\ 1 & \text{if } z>0 \end{cases} \tag{2.3}$$

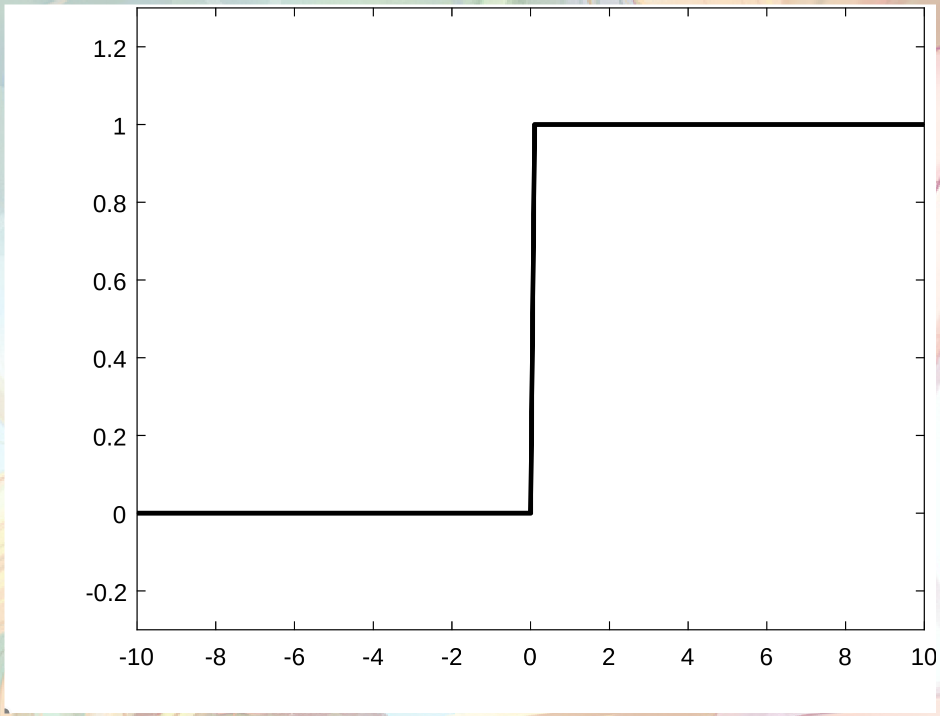


图 1: 感知器

可以直观地看到,在 $z=0$ 初函数值发生了突变,这就意味着函数无法在该点处求导,不具有良好的分析性质,为此,我们引入 S 神经元

$$f(\mathbf{x}) = \frac{1}{1+\exp(-z)} \tag{2.4}$$

可以看到,当 z 的值较大的时候,$f(\mathbf{x})\approx 1$,当 z 的值较小的时候,$f(\mathbf{x})\approx 0$,这与我们前面提出的感知器是一样的。而当 z 在零附近的时候,$f(\mathbf{x})$ 平滑地从 0 过渡到 1。由此,我们可以得出:S 神经元可以胜任感知器的大部分工作,并且具有良好的分析性质我们可以不妨选用这样一个 S 神经元作为感知器。

[1] $\mathbf{w}$ 和 b 是这个神经元的内在属性,因此自变量只有 $\mathbf{x}$

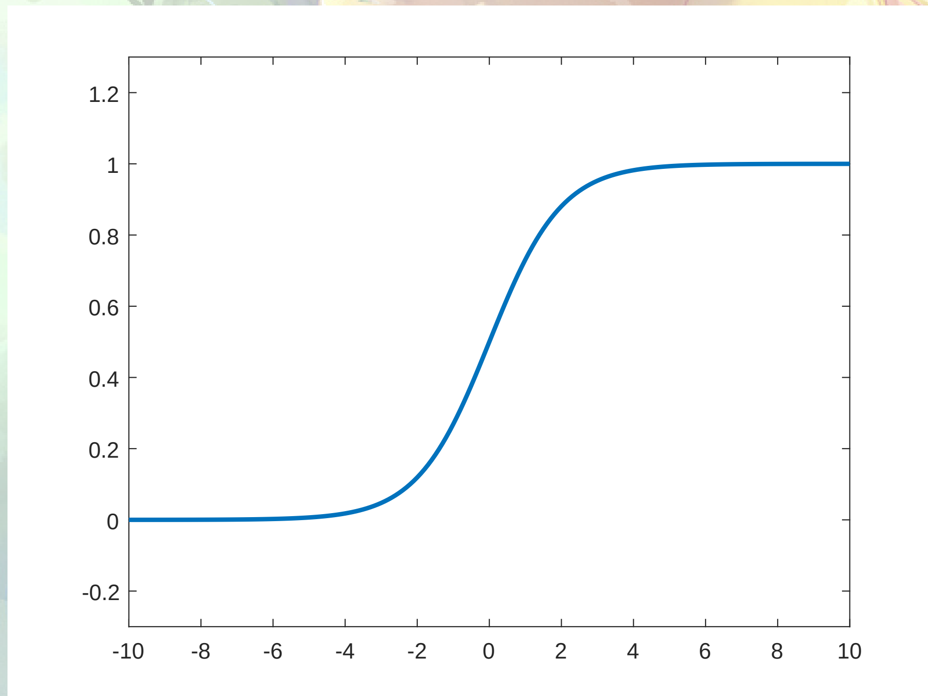


图 2: S 神经元

重新回到我们的神经网络, 输入层的神经元仅用作读取数据和传递数据, 不参与运算。而隐藏层的每一个神经元都要接收来自上一层的所有 x_i 并向下一层输出 $f(\mathbf{x})$。输出层的神经元也要接受来自上一层所有 x_i, 并给出一个输出结果, 结合所有输出层的神经元的输出结果, 我们就可以得到该神经网络的输出结果。

2.3 本例中的神经网络

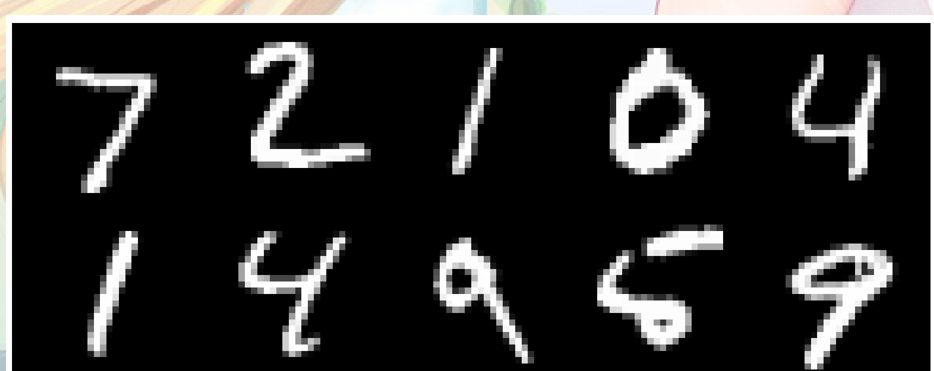


图 3: minst 数据集示意图

MNIST 数据集包含了 6 万张手写数字图片的训练集, 和 1 万的测试集, 每张规格为 28×28 大小的灰度图像[2]。经过简单的数据处理, 可以将图片转变为一个 784×1 的矩阵, 即将原 28×28 按列排列; 将标签转变为

[2] 在电子计算机领域中, 灰度(Gray scale)数字图像是每个像素只有一个采样颜色的图像。这类图像通常显示为从最暗黑色到最亮的白色的灰度, 尽管理论上这个采样可以是任何颜色的不同深浅, 甚至可以是不同亮度上的不同颜色。灰度图像与黑白图像不同, 在计算机图像领域中黑白图像只有黑白两种颜色, 灰度图像在黑色与白色之间还有许多级的颜色深度。

10×1 的向量,如果标签为 i ,则除第 $i+1$ 个元素为一外,其余所有元素都为零,例如:$[0,0,0,0,1,0,0,0,0,0]$ 表示这个图片的标签为 4。

基于此,输入层我们设计为 $28\times 28=784$ 个神经元,输出层为 10 个神经元。而中间层,这里我们设计成由 30 个神经元构成的只有一层的隐藏层,当然,我们这里也可以改成 50 个或者 10 个神经元,都没有太大的问题,但这会对后面的运算速度和结果产生些许的影响。

神经网络最终的输出结果要根据输出层的神经元的输出来判断,判断依据是,如果第 i 个神经元的输出值最大(后面我们会知道,这实际上是最接近于 1),那么我们就认为这个图像对应的数字是 i。

至于 b[3],w,这里我们采用随机生成的方式。具体来说,采用标准正态分布。

这样,我们就设计了一个三层的神经网络,第一层有 784 个神经元,第二层有 30 个神经元,第三层有 10 个神经元,初始参数均由标准正态分布随机数生成。

3 bp 神经网络的学习过程

让我们回想我们作为人类是如何学习认识数字的,当我们第一次接触数字的时候,我们也并不能说出它代表数字几,但我们在他人的帮助下,在一次次的失败中获得了经验,从而具有了判断能力。

神经网络也是一样,不难看出,对于一张确定的照片,影响输出结果的就是权重 ω 和偏置 b。于是我们下一步的方向就是,当神经网络犯错的时候,也就是做了错误分类的时候,如何引导神经网络选择正确的权重 ω 和偏置 b,从而纠正错误。

这样的学习方法有很多,在微积分中我们学到,函数值沿梯度的反方向下降速度最快,这里我们采用同样的思路。这引入了两个个新问题,如何定义函数,以及如何对函数求导。

为方便后面公式推导,这里给出符号说明:

符号	符号含义
a^l	第 l 层的激活值(输出值)矩阵
b^l	第 l 层的偏置矩阵
w^l	第 l 层的权重矩阵
a^l_j	第 l 层第 j 个神经元的激活值(输出值)
b^l_j	第 l 层第 j 个神经元的偏置
w^l_{jk}	连接第 $l-1$ 层的第 k 个神经元到第 l 层的第 j 个神经元的权重
$\odot$	Hadamard 乘积 $(s\odot t)_j=s_jt_j$
$\sigma(x)$	$\frac{1}{1+\exp(-x)}$

3.1 损失函数

损失函数是用于计算误差,评价模型预测结果是否准确的一个函数。因此,当模型的输出值与真实值非常接近的时候,损失函数应当接近零。为了便于后面求导,以及形式的简洁性,我们采用二次代价函数:

$$C=\frac{1}{2n}\sum_n\|y(x)-a^L(x)\|_2^2 \tag{3.1}$$

[3]可以看出,在 S 神经元中,b 不再代表感知器中那个突变的阈值,因此将其改名为偏置以作区别

其中 n 是训练样本的总数;求和运算遍历了每个训练样本 x;$y=y(x)$ 是对应的正确的目标输出;L 表示网络的层数;$a^L=a^L(x)$ 是当输入是 x 时的网络输出的激活值向量。

特别的,我们设

$$C_x=\frac{1}{2}||y(x)-a^L(x)||_2^2 \tag{3.2}$$

表示对于取定的一个输入值 x 的损失函数。

3.2 反向传播算法

我们定义 l 层的第 j 个神经元上的误差 δ_j^l 为

$$\delta_j^l=\frac{\partial C}{\partial z_j^l} \tag{3.3}$$

利用 δ 我们可以用下面的方程计算 $\frac{\partial C}{\partial b_j^l}$,$\frac{\partial C}{\partial w_{jk}^l}$

$$\delta^L=\nabla_a C\odot\sigma'\left(z^L\right) \tag{3.4a}$$

$$\delta^l=\left(\left(w^{l+1}\right)^T\delta^{l+1}\right)\odot\sigma'\left(z^l\right) \tag{3.4b}$$

$$\frac{\partial C}{\partial b_j^l}=\delta_j^l \tag{3.4c}$$

$$\frac{\partial C}{\partial w_{jk}^l}=a_k^{l-1}\delta_j^l \tag{3.4d}$$

下面给出上式的证明过程。

对于 (3.4a) 式:

证明.

$$\begin{aligned}\delta_j^L&=\frac{\partial C}{\partial z_j^L}\\&=\frac{\partial C}{\partial a_k^j}\frac{\partial a_k^j}{\partial z_j^L}\\&=\frac{\partial C}{\partial a_k^j}\sigma'(z^L)\end{aligned} \tag{3.5}$$

这就是 (3.4a) 式的分量形式 □

对于 (3.4b) 式:

证明.

$$\begin{aligned}\delta_j^l&=\frac{\partial C}{\partial z_j^l}\\&=\sum_k\frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l}\\&=\sum_k\frac{\sum_j w_{kj}^{l+1}\sigma\left(z_j^l\right)+b_k^{l+1}}{\partial z_j^l}\delta_k^{l+1}\\&=\sum_k w_{kj}^{l+1}\delta_k^{l+1}\sigma'\left(z_j^l\right)\end{aligned} \tag{3.6}$$

这就是 (3.4b) 式的分量形式 □

对于 (3.4c) 式:

证明.

$$\begin{aligned}\frac{\partial C}{\partial b_j^l} &= \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \\ &= delta_j^l\end{aligned} \tag{3.7}$$

这就是 (3.4c) 式的分量形式 □

对于 (3.4d)

证明.

$$\begin{aligned}\frac{\partial C}{\partial b_j^l} &= \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \\ &= \delta_j^l a_k^{l-1}\end{aligned} \tag{3.8}$$

这就是 (3.4d) 式的分量形式 □

3.3 沿梯度下降方向更新参数

利用 (3.4) 式,我们可以求出 $\frac{\partial C}{\partial b_j^l}, \frac{\partial C}{\partial w_{jk}^l}$。沿此方向,我们可以更新参数:

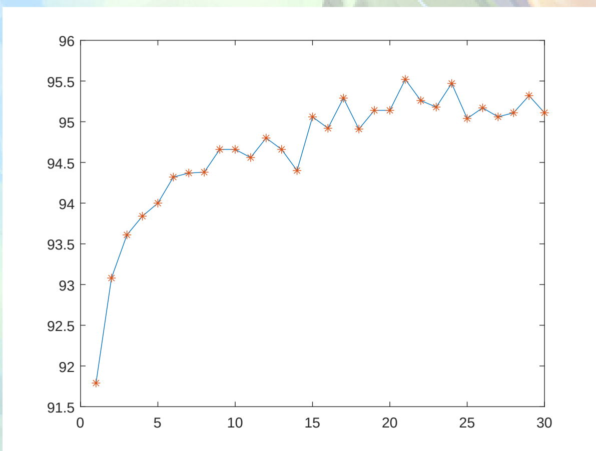
$$\begin{aligned}w_k \rightarrow w_k' &= w_k - \frac{\eta}{n}\sum_j \frac{\partial C_{X_j}}{\partial w_k} \\ b_l \rightarrow b_l' &= b_l - \frac{\eta}{n}\sum_j \frac{\partial C_{X_j}}{\partial b_l}\end{aligned} \tag{3.9}$$

其中 η 是学习率,如果学习率设置的过小,模型的求解速度会很慢,如果学习率设置的过大,模型就会难以找到最优解。

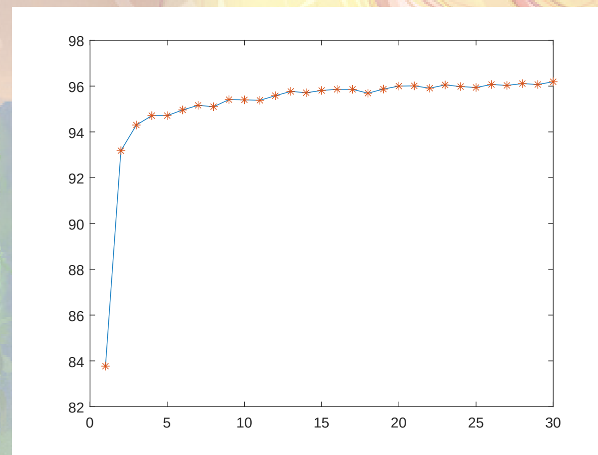
3.4 SGD 方法

观察 (3.9) 式可以发现,如果我们按照 (3.9) 式的方式来迭代,需要对所有的样本都进行运算一次 $\frac{\partial C_{X_j}}{\partial w_k}$, 和 $\frac{\partial C_{X_j}}{\partial b_l}$ 才能进行一次对 w, b 的更新。这就会导致需要很长的时间才能得到一组比较好的 w, b, 这个时间可能会超出我们的预期。

可以采用 SGD 方法来应对此问题:在一次大迭代中,我们重新打乱训练集的顺序,随机将训练集分成若干个小训练集,在每一个训练集上进行一次 (3.9) 式的迭代。如果我们记小训练集拥有的样本量为 m 个,那么就可以得到 n/m 个小训练集。也就是说,在每一次大迭代中,都可以进行 n/m 次参数更新,这大大促进了模型求解速度。



(a) 中间层设置为 30 个神经元



(b) 中间层设置为 50 个神经元

图 4: 模型准确率

4 利用神经网络来解决该数字分类问题

首先按照 2.3 中的方法生成一个神经网络。然后我们设置模型求解参数:迭代次数为 30,学习率 $\eta = 3$,小样本量为 10。

对每一次迭代结果,我们都做一次模型评估,将最终三十次迭代结果的准确性输出出来。

其中最好的一次达到了 95.52% 的准确率。把中间层设置为 50 个神经元,其余不变,输出结果如下:在改变参数后,最好的结果达到了 96.19%,并且准确率有着很好的稳定性,这意味着我们也许应当选择更多的神经元。

我们重点关注了一些识别错误的数据,发现我们的模型可能比看上去这个数字更好。右上角的标签是按照 NMIST 数据的正确的分类,而右下角的标签是我们组合网络的输出。应当承认的是,即便是人类面对这些图片也要费很大辛苦才能识别出来,甚至也会犯错。这意味着,我们的神经网络模型在 minst 上的能力已经在一定程度上接近人类水平了。

图 5: 分类错误的图片

5 神经网络的难以训练性

实际上, 本文设置的超参数都是经过精心调制的, 如果超参数设置的那么恰到好处, 模型的准确率会受到很大影响。可问题在于, 我们如何根据预先设置一个适当的超参数? 还是说只能通过不断的试错才能找到合适的超参数?

遗憾的是,实际情况可能是后者。

SGD 小样本的数量,学习率的大小,神经元的数量,激活函数的设置,损失函数的设置,甚至权重和偏置的初始化也会影响我们模型的正常运转。目前并没有一套完整的,令人信赖的流程来帮助我们设计网络,我们往往采用着直觉或者启发式的理由来进行这些参数的设置,当然,更多的时候还是要靠着不断试错。

A MATLAB 代码

main.m

```
clc,clear,close all

mnist_loader                    % 读取数据
sizes = [784, 30, 10];          % 输入层, 隐藏层, 输出层神经元数量
epochs = 7;                     % 迭代次数
mini_batch_size = 10;           % 小样本数量
eta = 3;                        % 学习率


%-------生成一个newwork---------%
biases = {randn(sizes(2),1),randn(sizes(3),1)}; % 偏置矩阵
weights = {randn(sizes(2),sizes(1)),randn(sizes(3),sizes(2))};  % 权重矩阵
[biases,weights] = net_SGD_train(sizes,biases,weights,train_x,train_y,test_x,test_y,epochs,mini_batch_size,3);

function  [biases,weights] = net_SGD_train(sizes,biases,weights,train_x,train_y,test_x,test_y,epochs,mini_batch_size,eta)
n = size(train_x,2);
for i = 1 : epochs
    randIndex_train = randperm(n);
    train_x = train_x(:,randIndex_train);
    train_y = train_y(:,randIndex_train);

    for j = 1:n/mini_batch_size
        mini_batchs_x = train_x(:,(j-1)*mini_batch_size+1:j*mini_batch_size);
        mini_batchs_y = train_y(:,(j-1)*mini_batch_size+1:j*mini_batch_size);
        [biases,weights] = update_mini_batch(mini_batchs_x,mini_batchs_y,eta,sizes,biases,weights);
    end
    correct_rate = evaluate(test_x,test_y,biases,weights);
    disp(strcat('第',num2str(i),'次迭代, 正确率为: ',num2str(correct_rate),'%'))
end
end
%---------------------------------------------------------------------%
%---------------------------------------------------------------------%
```

```
%-------------------------------------------------------------------------------%

function [biases,weights] = update_mini_batch(mini_batchs_x,mini_batchs_y, eta, sizes,biases,weights)

m = size(mini_batchs_x,2);
nabla_b = {zeros(sizes(2),1),zeros(sizes(3),1)};
nabla_w = {zeros(sizes(2),sizes(1)),zeros(sizes(3),sizes(2))};

for i = 1:m
    [delta_nabla_b,delta_nabla_w] = backprop(mini_batchs_x(:,i),mini_batchs_y(:,i),biases,weights);
    for j = 1:2
        nabla_b{j} = nabla_b{j} + delta_nabla_b{j};
        nabla_w{j} = nabla_w{j} + delta_nabla_w{j};
    end
end
for j = 1:2
    weights{j} = weights{j}-(eta/m)*nabla_w{j};
    biases{j} = biases{j}-(eta/m)*nabla_b{j};
end
end


%-------------------------------------------------------------------------------%
%-------------------------------------------------------------------------------%
%-------------------------------------------------------------------------------%

function [nabla_b, nabla_w] = backprop(x, y,biases,weights)

nabla_b = {[],[]};
nabla_w = {[],[]};


activation = x;
activations = {0,0,0};
zs = {0,0};
activations{1} = x;
% activations = [x] # list to store all the activations, layer by layer
```

```
% zs = [] # list to store all the z vectors, layer by layer
for i = 1:2
    z = weights{i} * activation + biases{i};
    zs{i} = z;
    activation = sigmoid(z);
    activations{i+1} = activation;
end

delta = cost_derivative(activations{3},y) .* sigmoid_prime(zs{2});
nabla_b{2} = delta;
nabla_w{2} = delta * transpose(activations{2});


z = zs{1};
sp = sigmoid_prime(z);

delta = transpose(weights{2}) * delta .* sp;
nabla_b{1} = delta;
nabla_w{1} = delta * transpose(activations{1});


end



%----------------------------------------------------------------------------%
%----------------------------------------------------------------------------%
%----------------------------------------------------------------------------%
function out = cost_derivative(output_activations, y)
out = output_activations - y;
end


%----------------------------------------------------------------------------%
%----------------------------------------------------------------------------%
%----------------------------------------------------------------------------%

function out = sigmoid(z)
out = 1./(1+exp(-z));
```

```
end

%---------------------------------------------------------------------------%
%---------------------------------------------------------------------------%
%---------------------------------------------------------------------------%
function out = sigmoid_prime(z)
out = sigmoid(z).*(1-sigmoid(z));
end

%---------------------------------------------------------------------------%
%---------------------------------------------------------------------------%
%---------------------------------------------------------------------------%
function a = feedforward(a,biases,weights)

for i = 1:2
    b = biases{i};
    w = weights{i};
    a = sigmoid(w*a+b);
end
end


%---------------------------------------------------------------------------%
%---------------------------------------------------------------------------%
%---------------------------------------------------------------------------%

function correct_rate = evaluate(test_x,test_y,biases,weights)
n = size(test_x,2);
correct = 0;
for i = 1:n
    y1 = feedforward(test_x(:,i),biases,weights);
    [~,y1] = max(y1);
    y2 = test_y(:,i);
    [~,y2] = max(y2);
    if y1==y2
        correct = correct + 1;
    end
end

```

```
correct_rate = correct/n*100;
end
```